L3 - Understanding Class Definitions

February 11, 2020

1 Understanding Class Definitions - Exploring Source Code

The code below explores the behaviour of a simple ticket machine.

Interacting with the `TicketMachine()` object gives us information about its internal functionality.

1.1 Keywords

The keywords are also known as *reserved words* and cannot be used as names. They are always entirely lowercase.

1.1.1 `public` and `private`

The visibility modififers. If set to `private`, only methods within the object can use this state variable. Conversely, if `public`, method calls from outside the object can access this variable.

1.1.2 `class`

Provides for a class declaration.

1.2 Fields

Also *instance variables*. These define the state of an object. In BlueJ, use *Inspect* to view all object state.

1.3 Constructors

Constuctors set up an object upon initialization. They always have the same name as the class. Usually stores initial values into the state variables; often from paramaters passed to the constructor.

In Java, arguments are always passed by *value*.

1.4 Assignment

Stores the value on the RHS inot the variable on the LHS. Don't confuse with the equality test. Upon assinment, previous values of the variable are lost.

1.5 Methods

Methods impement the behaviour (rather than the state) of objects. Methods have *headers* and *bodies*: the header, eg. `public int getPrice();`, tells us **visibility**, what (if anything) the method **returns**, the **name** of the method, and what (if any) **parameters** the method takes.

1.5.1 Accessors

AKA *getter methods*. Return the current state (or some detail thereof) to a calling procedure. For example:

```
public int getPrice();
{
    return price;
}
```

In this example, the return type is `int`, the visiblilty is `public`, and the parameter list is empty. For the accessor to be of use, it must have a non-`void` return type.

Accesing a parameter is *not* the same as printing a parameter.

1.5.2 Mutators

AKA *setter methods*. *Modify* the current state of the object.

```
public void insertMoney(int amount)
{
    balance = balance + amount;
}
```

Generally, we will want to pass parameters to mutator methods. It will generally also have at least one assignment statement in its body.

`set` methods We will often want to have dedicated `set` mutator methods. These have a simple form:

- `void` return type
- a single parameter
- a name including a field name

Protective Mutators A method does not need to unquestioningly assign a value - we could use `if` or `else` to reject invalid values. We can thereby protect fields from nonsense: this is an example of *encapsulation* in OOP.

1.6 Broken Code

[ ]:
```
// find 5 errors in this code

public class CokeMachine
{
    private price; // no type
```

```
    public CokeMachine() // no return type
    {
        price = 300 // no semicolon
    }

    public int getPrice // no brackets
    {
        return Price; // wrong case
    }
}
```

1.7 Fixed Code

[16]:
```
// fixed!

public class CokeMachine
{
    private int price;

    public void CokeMachine()
    {
        price = 300;
    }

    public int getPrice()
    {
        return price;
    }
}
```

1.8 Naive Ticket Machine

Note that in the `printTicket()` method we show that the `+` operator is *overloaded*: it will do different things depending on what data we ask it to operate on:

- `String` + `String`: string concatenation
- `int` + `int`: addition
- `String` + `int` + `String`: concatenation again
- `int` + `int` + `String`: arithmetic addition, *then* string concatenation
  - fml

Operator precedence therefore goes from left to right:

- `int` + `int` = integer sum, a new `int`
- then it's `int` + `string`, which is a concatenation

[1]:
```java
// ctrl-v ctrl-c from the naive-ticket-machine example

/**
 * TicketMachine models a naive ticket machine that issues
 * flat-fare tickets.
 * The price of a ticket is specified via the constructor.
 * It is a naive machine in the sense that it trusts its users
 * to insert enough money before trying to print a ticket.
 * It also assumes that users enter sensible amounts.
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2016.02.29
 */
public class TicketMachine // the outer wrapper of TicketMachine
{
    // the constructor for TicketMachine
    // The price of a ticket from this machine.
    private int price;
    // The amount of money entered by a customer so far.
    private int balance;
    // The total amount of money collected by this machine.
    private int total;

    /**
     * Create a machine that issues tickets of the given price.
     * Note that the price must be greater than zero, and there
     * are no checks to ensure this.
     */
    public TicketMachine(int cost)
    {
        price = cost;
        balance = 0;
        total = 0;
    }

    /**
     * Return the price of a ticket.
     */
    public int getPrice()
    {
        return price;
    }

    /**
     * Return the amount of money already inserted for the
     * next ticket.
     */
```

```java
    public int getBalance()
    {
        return balance;
    }

    /**
     * Receive an amount of money from a customer.
     */
    public void insertMoney(int amount)
    {
        balance = balance + amount;
    }

    /**
     * Print a ticket.
     * Update the total collected and
     * reduce the balance to zero.
     */
    public void printTicket()
    {
        // Simulate the printing of a ticket.
        System.out.println("##################");
        System.out.println("# The BlueJ Line");
        System.out.println("# Ticket");
        System.out.println("# " + price + " cents.");
        System.out.println("##################");
        System.out.println();

        // Update the total collected with the balance.
        total = total + balance;
        // Clear the balance.
        balance = 0;
    }
}
```

[13]:

```java
// as if THIS is the syntax for a new object, smdh

TicketMachine tm = new TicketMachine(1);
```

[14]:

```java
tm.getPrice();
```

[14]: 1

[17]:

```java
tm.insertMoney(2); // return type is void
```

[18]:

```java
tm.getBalance();
```

[18]: 2

[19]:
```
tm.printTicket();
```

```
##################
# The BlueJ Line
# Ticket
# 1 cents.
##################
```

[20]:
```
tm.getBalance()
```

[20]: 0

Well this is obviously a terrible ticket machine given that it literally cannot *subtract two numbers correctly*.

1.9 Summary

- Methods implement all object behaviour.
- A method has a name and a return type at minimum.
  - The return type may be void

1.10 The Better Ticket Machine

It would be neat if the ticket machine could do basic math:

- check if the money amount is sensible
- ensure enough money to pay for ticket before printing ticket
- gives change if too much money put in the machine

[21]:
```
/**
 * TicketMachine models a ticket machine that issues
 * flat-fare tickets.
 * The price of a ticket is specified via the constructor.
 * Instances will check to ensure that a user only enters
 * sensible amounts of money, and will only print a ticket
 * if enough money has been input.
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2016.02.29
 */
public class BtrTicketMachine
{
    // The price of a ticket from this machine.
    private int price;
```

```java
    // The amount of money entered by a customer so far.
    private int balance;
    // The total amount of money collected by this machine.
    private int total;

    /**
     * Create a machine that issues tickets of the given price.
     */
    public BtrTicketMachine(int cost)
    {
        price = cost;
        balance = 0;
        total = 0;
    }

    /**
     * @Return The price of a ticket.
     */
    public int getPrice()
    {
        return price;
    }

    /**
     * Return The amount of money already inserted for the
     * next ticket.
     */
    public int getBalance()
    {
        return balance;
    }

    /**
     * Receive an amount of money from a customer.
     * Check that the amount is sensible.
     */
    public void insertMoney(int amount)
    {
        if(amount > 0) {
            balance = balance + amount;
        }
        else {
            System.out.println("Use a positive amount rather than: " +
                               amount);
        }
    }
```

```java
    /**
     * Print a ticket if enough money has been inserted, and
     * reduce the current balance by the ticket price. Print
     * an error message if more money is required.
     */
    public void printTicket()
    {
        if(balance >= price) {
            // Simulate the printing of a ticket.
            System.out.println("##################");
            System.out.println("# The BlueJ Line");
            System.out.println("# Ticket");
            System.out.println("# " + price + " cents.");
            System.out.println("##################");
            System.out.println();

            // Update the total collected with the price.
            total = total + price;
            // Reduce the balance by the price.
            balance = balance - price;
        }
        else {
            System.out.println("You must insert at least: " +
                               (price - balance) + " more cents.");

        }
    }

    /**
     * Return the money in the balance.
     * The balance is cleared.
     */
    public int refundBalance()
    {
        int amountToRefund;
        amountToRefund = balance;
        balance = 0;
        return amountToRefund;
    }
}
```

[22]:
```java
BtrTicketMachine btm = new BtrTicketMachine(55); // still a stupid syntax
```

[23]:
```java
btm.getPrice()
```

[23]: 55

[24]:
```
btm.insertMoney(100)
```

[25]:
```
btm.getBalance()
```

[25]: 100

[26]:
```
btm.printTicket()
```

```
##################
# The BlueJ Line
# Ticket
# 55 cents.
##################
```

[28]:
```
btm.getBalance()
```

[28]: 45

[29]:
```
btm.refundBalance()
```

[29]: 45

[30]:
```
btm.getBalance()
```

[30]: 0

That seems like a more reasonable ticket machine tbh.

1.11 Conditional Statements

If I have enough money, I will get a pizza. *Else*, I will stay home and watch Netflix:

```
if ( me.haveMoney() )
{
    restaurant.getPizza();
}
else
{
    me.watchNeflixSadly();
}
```

These conditional statements takes a value as a result of a test. **Boolean** expressions have two possible values: *true* and *false*.

For example, in the `insertMoney` method in `BtrTicketMachine`, the `amount > 0` avoids a nonsense action for garbage input.